Processes_definition

Программа – совокупность файлов
(исходных, объектных или, собственно, исполняемый файл).
Для запуска надо создать ENVIRONMENT окружение, среду выполнения.

Процесс – программа на стадии ее выполнения, состоит из:

- инструкций, выполняемых процессором
- текущих значений регистров и счетчиков
- текущих значений переменных
- информации о выполняемой задаче:
  - размещение в памяти
  - открытые файлы
  - статус процесса и др.

Processes_interaction

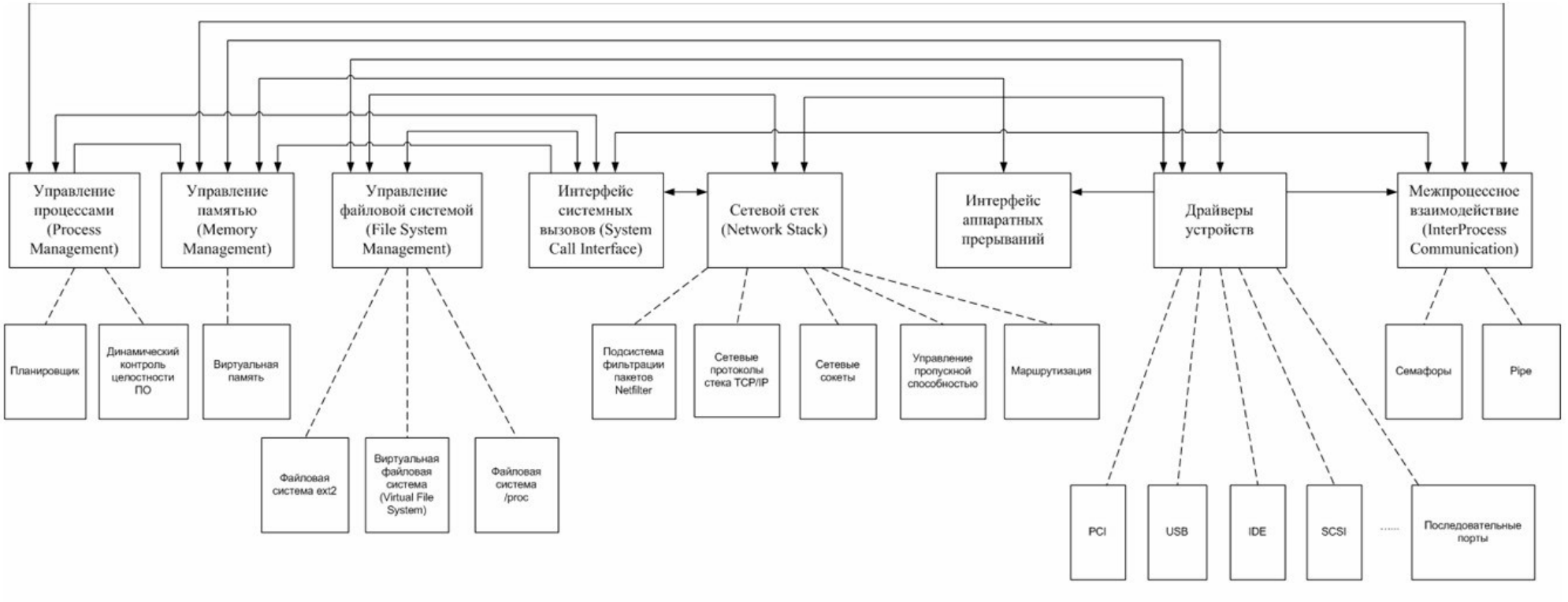
Процессы **изолированы** друг от друга (каждый следует собственному набору инструкций, использует свой сегмент данных, стек и др.).

Для обмена данными и синхронизации процессов в Linux существуют средства, образующие систему межпроцессного взаимодействия

IPC InterProcess Communication facilitiues :

- (pipes) каналы
- (message queue) очереди сообщений
- (semaphores) семафоры
- (signals) сигналы
- (shared memory) разделяемая память
- (sockets) сокеты

Kernel_subsystems interaction



Processes_types&hierarchy

Системные процессы – часть ядра системы:

- запускаются в определенном порядке при инициализации ядра
- всегда расположены в оперативной памяти
- их инструкции и данные находятся в области ядра
- вызывают функции и обращаются к данным, недоступным остальным

Прародителем всех процессов в Linux является процесс

init,

Все процессы принадлежат единому дереву, начинающемуся с init.
Имеется четкая иерархия процессов. Предопределенная цепочка запусков.

Processes_init&daemons

- Запуск процесса **init** происходит из исполняемого файла etc/init.
- **Init** читает файл etc/inittab с инструкциями для дальнейшей работы.
- Первая инструкция - запуск скрипта инициализации etc/initd.
- Здесь произойдет проверка и монтирование файловых систем, установка часов системного времени, включение своп-раздела, присвоение имени хоста и т.д.
- Вызывается следующий скрипт, приводящий к запуску предопределенного набора демонов.

Демоны

– это неинтерактивные процессы, выполняющиеся в фоновом ***background*** режиме и они обычно:

- не связаны ни с одним пользователем (сеансом работы),
- большую часть времени ожидают пока какой-либо процесс запросит услугу (печать, доступ к файлу и т.д.).

Processes_daemons examples

Некоторые из процессов-демонов:

- **crond** – периодически читает общесистемное расписание из /etc/crontab и запускает команды (процессы) при наступлении момента времени.
- **syslogd** – отвечает за систему журнальной регистрации событий, записывает системные сообщения в файлы журналов /var/log.
- **xinetd** – управляет сервисами для Интернета, прослушивает сокеты и, если в каком-либо есть сообщение, вызывает соответствующую программу для обработки запроса.

Processes_launch completion

- В зависимости от сложности ядра и вида дистрибутива Linux имена и количество (сотни) запускаемых при загрузке ОС процессов разнятся.
- На завершающей стадии запуска системы:
  - Запускается процесс управляющий виртуальным терминалом **tty**, назначением, которых является слежение за консолями пользователей.
  - Процесс **login** ожидает начала сеанса работы пользователя в системе.
  - После успешной регистрации пользователя запускается **shell** оболочка-командный интерпретатор (**bash** или др.), обеспечивающий работу пользователя (обработка команд пользователя, запуск процессов).
- Окончание сеанса **LogOut** приводит к завершению работы **shell** и отключению пользователя от системы.

Processes_application processes

Прикладные процессы

- Порождаются в рамках пользовательского сеанса работы.
- Могут выполняться, как в интерактивном **foreground** так и в фоновом **background** режимах.
- Интерактивные процессы монопольно владеют терминалом, с которого запущены.
- Пока интерактивный процесс не завершится, пользователь не запустит другой (только с другого терминала).
- Время жизни (и выполнения) прикладного процесса ограничено сеансом работы пользователя.

Processes_attributes

Атрибуты процессов

- **PID** – уникальный идентификатор процесса Process ID.
- **PPID** – идентификатор породившего процесса Parent Process ID.
- **UID** – реальный идентификатор ID пользователя User ID запустившего процесс.
- **EUID** – эффективный идентификатор Effective User ID, для опр-ния прав доступа.
- Обычно процесс имеет те же права, что и пользователь (UID~EUID), но их можно расширить, например, установкой флага SUID (идентификатору EUID присвоится значение идентификатора владельца исполняемого файла, администратора.)
- **GID** – реальный идентификатор группы запустившего процесс пользователя.
- **EGID** – эффективный идентификатор группы запустившего процесс пользователя.

Processes_status

Состояния процесса

Для управления процессами ОС содержит таблицу процессов с информацией об их состоянии, атрибутах и др.

Processes_table

Таблица процессов

Управление процессом	Управление памятью	Управление файлами
Регистры	Указатель на текстовый сегмент	Корневой каталог
Счетчик команд	Указатель на сегмент данных	Рабочий каталог
Слово состояния программы	Указатель на сегмент стека	Дескрипторы файла
Указатель стека		Идентификатор пользователя
Состояние процесса		Идентификатор группы
Приоритет		
Параметры планирования		
Идентификатор процесса		
Родительский процесс		
Группа процесса		
Сигналы		
Время начала процесса		
Использованное процессорнсе время		
Процессорное время дочернего процесса		
Время следующего аварийного сигнала		

Processes_useful commands

- **ps** - Вывести список процессов
- **top** - Интерактивно наблюдать за процессами
- **uptime** - Посмотреть загрузку системы
- **free** - Вывести объем свободной памяти
- **pstree** - Отображает все запущенные процессы в виде иерархии

Processes_creation&termination

Создание (порождение) процесса, системные вызовы

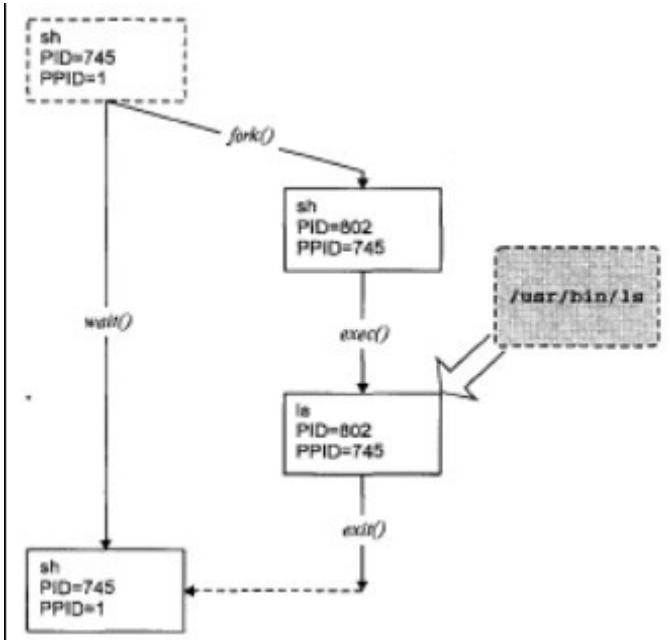
fork() и exec()

#include <sys/types.h>

#include <unistd.h>

…

pid_t fork(void);



Processes_creation&termination fork()

Возвращаемое значение fork() позволяет определить, где находится управление

```
. . .
int pid;
pid = fork();
if (pid == -1) {
        perror("fork"); exit (1);
}

If (pid == 0) {
/* Эта часть кода выполняется дочерним процессом */
                printf ("процесс-потомок\n");
}
Else            {
/* Эта часть кода выполняется родительским процессом */
printf ("процесс-родитель\n");
        }
. . .
```

Processes_creation&termination inheritance

Порожденный процесс (потомок, дочерний..) является точной копией **родительского** (его клоном).

Потомок **наследует** такие атрибуты процесса-родителя, как

- Идентификаторы пользователя **UID** и группы **GID**.
- Переменные окружения **environment variables**.
- Дескрипторы файлов.
- Текущий каталог.
- Управляющий терминал.
- Диспозицию сигналов и их обработчиков.
- Ограничения процесса **rlimits**.

Образ виртуальной памяти потомка повторяет родительскую, такие же сегменты кода, данных и стека (code, data, stack).

Processes_creation&termination fork()

Процесс-потомок и процесс-родитель имеют **отличия**:

- Потомку присваивается новый уникальный идентификатор **PID**.
- Идентификаторы **PPID** разные.
- Системный вызов fork() возвращает разные значения:
  - процессу родителю - значение **PID** процесса потомка (-1 => fault),
  - процессу потомку - значение равное **0**.

Это позволяет определить, кто является потомком, а кто родителем, и разделить в дальнейшем функциональность этих процессов, который будут выполняться параллельно.

Processes_clone asynchronous

```
/* Программа forkdemo.cpp */
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
main()
{
int i;
   if (fork()) { /* Это процесс-родитель */
         for(i=0; i<1000 ; i++)
           printf("\t\tPARENT %d\n", i);
         }
        else { /* Это процесс-потомок */
          for(i=0; i<1000 ; i++)
              printf("CHILD %d\n",i);
           }
}
```

Processes_creation&termination exec

Системный вызов **exec()**

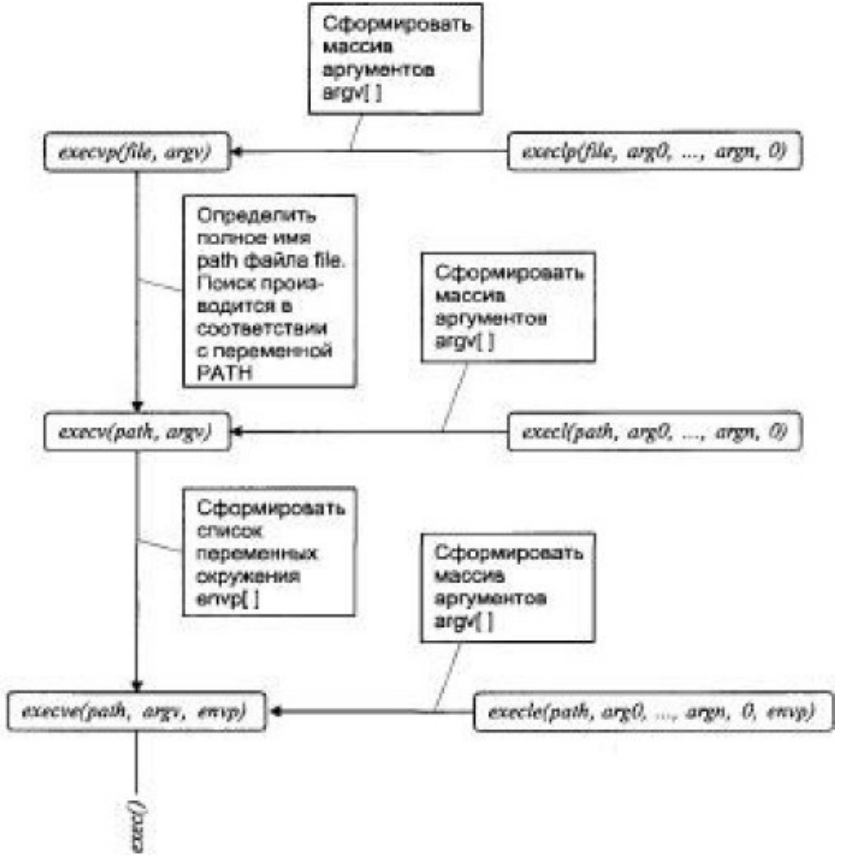
загружает другой исполняемый (смысловой) файл из процесса-потомка.

Образ существующего процесса-потомка полностью заменяется на загружаемый из файла, указанного при вызове **exec()**.
Загружаемая программа наследует от процесса-потомка:

- идентификаторы процесса **PID** и **PPID**.
- идентификаторы пользователя и группы;
- дескрипторы файлов, для которых не установлен флаг FD CLOEXE;
- текущий каталог;
- управляющий терминал;
- ограничения процесса **rlimits**.

Модификации **exec()** отличаются постфиксом мнемоник (6 видов).

Processes_creation&termination exec types



Processes_creation&termination tinymenu

В программе *tinymenu.cpp* с помощью системного вызова **execlp()**, передающего параметры командной строки списком и наследующего переменные окружения,запускаются команды интерпретатора **shell** .
После отработки соответствующей команды **shell** управление обратно, само по себе не возвращается.

```
/* Программа tinymenu.cpp */
#include<stdio.h>
#include<unistd.h>
main()
{
        /* Фиксированный список команд */
static char *cmd[]={"who","ls","date"};
int i;
        /* Подсказка номера команды */
printf("0=who, 1=ls, 2=date:");
scanf("%d",&i);
        /* Запуск выбранной команды на исполнение */
   execlp(cmd[i], cmd[i], 0);
printf("Command not found\n");
        /* запуск не удачный */
}
```

Thanks for your attention

Спасибо за внимание !

vladimir.shmakov.2012@gmail.com